Symbolic Execution Based Automatic Performance Optimization for MPI Programs

Zheng Bian
State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology
National University of Defense Technology
Changsha, Hunan, China
bianzheng22@nudt.edu.cn

Zhenbang Chen*
State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology
National University of Defense Technology
Changsha, Hunan, China
zbchen@nudt.edu.cn

Ji Wang
State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology
National University of Defense Technology
Changsha, Hunan, China
wj@nudt.edu.cn

Abstract

Overlapping communication with computation is a common way to optimize parallel programs in high-performance computing. However, this optimization is often carried out manually and is challenging to be automated to achieve a better performance. In this extended abstract, we report our recent progress of a symbolic execution-based automatic method for optimizing Message Passing Interface (MPI) programs by overlapping communication and computation. The key idea is to transform blocking communication operations into non-blocking ones while preserving the correctness of computation. We employ symbolic execution to extract the symbolic paths from the MPI program, based on which we can find the potential locations for optimization in each path. Then, we synthesize a global optimization transformation based on the potential optimization locations. The MPI program's performance can be improved after applying the global optimization transformation. The preliminary experimental results indicate the promise of our method.

CCS Concepts

• **Software and its engineering → Software performance**; **Automated static analysis**; **Access protection**.

Keywords

MPI, Perfomance Optimization, Symbolic Execution

ACM Reference Format:
Zheng Bian, Zhenbang Chen, and Ji Wang. 2025. Symbolic Execution Based Automatic Performance Optimization for MPI Programs. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3696630.3731438

*Corresponding Author

FSE Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3731438

1 Introduction

Message Passing Interface (MPI) is the de-facto programming standard in high-performance computing (HPC), but communication latency often dominates the execution time of communication intensive MPI programs. Non-blocking operations (*e.g.*, `MPI_Isend`) allow communication to overlap with computation, hiding latency. However, overlapping may cause faults if data is modified before being sent. To ensure correctness, `MPI_Wait` blocks execution until the communication completes, typically placed before accessing related data. The code between a non-blocking call and its wait is the *overlapping window*, where communication and computation run in parallel. A larger overlapping window can enhance performance.

Manually overlapping communication and computation is common but tedious and error-prone when optimizing legacy blocking codes. It is desirable and challenging to automatically transform blocking communications to non-blocking ones, including automatically inserting the wait operations to preserve the computation's correctness while targeting the largest overlapping window. Since correctness depends on data dependencies, program analysis techniques can be used to automate this transformation and enhance performance.

Automatic optimization approaches are either static [1, 3, 6] or dynamic [4]. Static methods analyze memory accesses to ensure correctness, but limited precision often misses the largest overlapping window. Dynamic methods can find optimal overlaps per path but suffer from input coverage gaps and runtime overhead. Achieving maximal overlap with low overhead and guaranteed correctness remains challenging.

Symbolic execution is a precise analysis method that balances dynamic and static analysis methods, which covers the input space while preserving the analysis precision. Though time-consuming, it is suitable for compile-time use to optimize HPC programs, where runtime performance is critical.Based on this insight, we propose to employ symbolic execution to find the largest overlapping windows for the MPI program. In this paper, we present our in-progress work, called MPI-SO, which uses symbolic execution to precisely capture the memory operations related to communications in the program and automatically transform blocking communications to non-blocking ones. The key idea is to identify the path-level overlapping windows by symbolic execution and synthesize a global optimization transformation. Built on an existing MPI symbolic execution framework [2], MPI-SO's preliminary results show that it

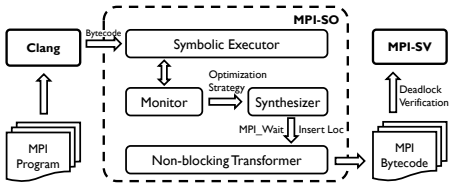

Figure 1: MPI-SO Framework and Workflow

creates larger overlapping windows than static methods by crossing loops and function calls.

2 Symbolic Optimizer Framework

Figure 1 illustrates MPI-SO's framework and workflow. We symbolically execute MPI bytecode compiled by clang, during which the **Monitor** observes communication operation related memory ranges. Memory operation on these ranges are handed over to the **Synthesizer**. After synthesize, a unified optimization plan will be adopted by **Non-blocking Transformer** and perform transformation on the bytecode. To address potential deadlocks from increased asynchrony, we verify the transformed program using MPI-SV [2]. In the following chapters, we will introduce the elements depicted in the figure.

2.1 Monitor

The monitor identifies memory operations associated with each communication operation within the current symbolic execution path via assuming that the communication operations are in their corresponding non-blocking ones.

Once a memory operation accesses the memory used by a communication operation, which triggers a data error, the monitor can identify an *optimization strategy* for the current execution path.

2.2 Synthesizer

Due to loops in the program and the multi-path exploration in symbolic execution, a single communication operation may have multiple potential optimization strategies. For correctness, we need to synthesize different optimization strategies globally to derive a unified wait operation insert location to create overlapping windows, which is the *nearest common pre-dominator* of the memory operations in different optimization strategies.

2.3 Non-blocking Transformer

Communication operations will be transformed to non-blocking ones according to their insertion locations determined by during global synthesis. We will create `MPI_Request` and `MPI_Status` to insert wait operations, in addition to some auxiliary flags. Our implementation leverages an LLVM Pass to facilitate the transformation.

3 Preliminary Experiment

The prototype of MPI-SO is now capable of transforming all blocking communication operations into non-blocking ones. Previous experiments [1] have shown that program execution time is not a reliable metric for evaluating the effectiveness of overlapping window. In addition to using **the number of instructions in overlapping windows** as a static metric [5], we propose a dynamic metric—**the execution time of overlapping windows**—to mitigate the limitations caused by varying execution costs of different instruction types. This new metric more accurately reflects the actual size of overlapping windows during program execution. In our preliminary experiments, MPI-SO showed good performance on both metrics.

```
1   do{
2     /*...Code Omitted...*/
3     if(flag[0]){
4       MPI_Wait(&req[0], &stat[0]);
5       flag[0] = 0;
6     }
7     for(...){
8       Bloc_Vector_X[index] = ...;
9       /*...Code Omitted...*/
10    }
11    MPI_Iallgather(Bloc_Vector_X, Bloc_VectorSize,
          MPI_DOUBLE, ..., &req[0]);
12    flag[0] = 1;
13    /*...Code Omitted...*/
14  } while(...);
```

Listing 1: congrad.c after transform

In Listing 1, MPI-SO can transcend loop body boundaries and insert the wait operation of Line 11's collective communication just before Line 7, *i.e.*, the subsequent loop iterations. The number of instructions in the overlapping window MPI-SO created is 2x of that using Petal[1]. Besides, under 12 running configurations, the execution time of MPI-SO's overlapping window is on average 24x of that using Petal. These results indicate the effectivness of MPI-SO.

4 Next Step

Our future work will be mainly carried out in the following aspects: (1) Evaluate MPI-SO on more benchmarks; (2) Instead of applying an unified wait operation, we directly apply optimization strategies of different execution paths, thereby maximizing the overlapping window for explored paths.

Acknowledgments

This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Programs (No. 62172429 and 62032024).

References

[1] Hadia Ahmed and et.al. 2017. Transforming blocking MPI collectives to Non-blocking and persistent operations. In *EuroMPI '17*. Chicago, IL, USA, September 25–28, 2017, 11 pages. https://doi.org/10.1145/3127024.3127033

[2] Zhenbang Chen and et.al. 2020. MPI-SV: a symbolic verifier for MPI programs. In *ICSE '20 (2020)*. 93–96. https://doi.org/10.1145/3377812.3382144

[3] Anthony Danalis and et.al. 2009. MPI-aware compiler optimizations for improving communication-computation overlap. In *ICS '09 (2009)*. 316–325. https://doi.org/10.1145/1542275.1542321

[4] Alexis Lescouet and et.al. 2020. Transparent Overlapping of Blocking Communication in MPI Applications. In *HPCC/SmartCity/DSS (2020)*. IEEE, 744–749. https://doi.org/10.1109/HPCC-SmartCity-DSS50907.2020.00097

[5] Van Man Nguyen. 2022. *Compile-time Validation and Optimization of MPI Non-blocking Communications*. Ph. D. Dissertation. Bordeaux. https://theses.fr/2022BORD0415

[6] Van Man Nguyen and et.al. 2020. Automatic Code Motion to Extend MPI Nonblocking Overlap Window. In *High Performance Computing*. 43–54. https://doi.org/10.1007/978-3-030-59851-8_4

Received 2025-2-28; accepted 2025-4-10